Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Electronic Notes in Theoretical Computer Science 133 (2005) 61–79
Electronic Notes in Theoretical Computer Science
www.elsevier.com/locate/entcs

Model Checking Flight Guidance Systems: from Synchrony to Asynchrony

Yunja Choi

Fraunhofer Institute for Experimental Software Engineering
Kaiserslautern, Germany

Abstract

Model checking has become a promising automated verification technique in practice. Nevertheless, most existing model checkers are specialized for limited aspects of a system where each of them requires a certain level of expertise to use the tool in the right domain in the right way. Hardly any guideline is available on choosing the right model checker for a particular problem domain, which makes adopting the technique more difficult in practice.
Based on the author's prior experience with the use of the symbolic model checker NuSMV on commercial Flight Guidance Systems (FGS) at Rockwell-Collins, the relative benefits and pitfalls of using the explicit model checker SPIN on the same problem are investigated. This has been a question from the beginning of the project with Rockwell-Collins. The challenge includes an efficient use of SPIN for the complex synchronous mode logic with a large number of state variables, where SPIN is known to be not particulary efficient. We present the way the SPIN model is optimized to avoid the state space explosion problem, which makes SPIN scale up better than NuSMV in the end, and discuss the implication of the result. We hope our experience can be a useful reference for the future use of model checking in a similar domain.

Keywords: Model Checking, Flight Guidance Systems, SPIN v.s. NuSMV

1 Introduction

Model checking [4,15] has become a promising technique for automated verification of software and hardware systems. Motivated by a few success stories of applying the technique in practice [2,9,10,12,14,21], the Critical Systems Research Group at the University of Minnesota has integrated the symbolic model checker NuSMV [22] into the specification-centered system development

[1] Email:choi@iese.fraunhofer.de

doi:10.1016/j.entcs.2004.08.058

environment Nimbus [13]. The integrated model checker has been successfully used for checking hundreds of requirements properties of specifications of commercial Flight Guidance System(FGS), a component of the Flight Control System, at Rockwell-Collins [3,21]. The success of the project is particulary meaningful to formal methods in practice since all the requirements engineering activities for the FGS, from writing specifications to conducting formal verifications, are performed by the practitioners at Rockwell-Collins.

Nevertheless, the choice of the right model checker has been a question since the beginning of the project; comparative studies on various model checkers in the application domain are quite rare, and, thus, the decision had to be based on anecdotal case studies. Despite the success of the project, we have been wondering whether other choices would have been better. Especially, the choice between symbolic and explicit model checking is quite unclear, with only limited comparative arguments; among others, it has been argued that symbolic model checking performs better for synchronous systems with hardware-like characteristics and explicit model checking is better for asynchronous systems with a number of communicating processes. Many reports have pointed out, however, that a direct comparison of the two techniques is very difficult, if not impossible [1,7,16]. Some experiments report that a symbolic model checker performs better even for asynchronous systems [7], contradicting some of the arguments.

Furthermore, a system can have both synchronous and asynchronous aspects depending on which part we are interested in and how we interpret the system behavior. For example, the FGS system consists of two FGSs, an active FGS and a passive FGS for back-up, running in parallel. The system is an asynchronous system when we consider its two communicating processes running in parallel; it can be interpreted as a synchronous system, with the complex mode logic complex enough to challenge the capability of the model checking technique, when we focus only on the mode logic of one-side FGS.

We have investigated the use of SPIN [17] – a representative explicit model checker – on FGS to get a better understanding of the capabilities of explicit versus symbolic model checking in our domain. We present the application of the SPIN model checker on the specifications of FGSs and compare the result to the one with NuSMV we have reported earlier [3,21]. This work involves the challenge of translating a synchronous specification into a modelling language designed for specifying communicating processes. Our direct translation shows a disastrous performance with SPIN, quickly blowing up on even a small-size FGS. The problem is mainly due to the large number of global and local variables, including history values, accessed by both the system model and its environment model. After a careful review, we encapsulated as many variables

as possible within each process and used the message passing mechanism to allow other processes to access the values of the shared variables. The result is quite encouraging; SPIN is able to check a property on an FGS model within a reasonable amount of time. Furthermore, the same approach makes it possible to verify an important property regarding two communicating FGSs — the same property NuSMV fails to scale up to check.

Our optimization is systematic and performs minimal changes in order to support automation of the verification process. Considering the performance gain from this minimal optimization, we believe there is much more room for further performance improvement. In other words, SPIN can be more flexible than NuSMV in handling large scale systems, but can also be more difficult to use by non-experts, as optimization can raise a couple of issues in terms of usability: (1) Property-based optimization requires different models for different properties, which can be an issue when checking hundreds of requirements properties, and (2) more aggressive optimization may improve performance, but may result in dramatic changes of the original model, which makes it difficult for the modeler to understand. It is our belief that any optimization approach must be systematic so that it can be automated, providing traceability between the original and the optimized model. This is the focus of our future investigation.

The remainder of this paper is organized as follows: Section 2 discusses existing related work focusing on the comparison of symbolic and explicit model checking techniques. Section 3 introduces our motivation with a brief description of the existing work related to model checking Flight Guidance Systems. Section 4 describes our direct and modular translations to the SPIN model with their performance data. We conclude with a discussion about the implication of the result in Section 5.

2 Related Work

The pros and cons of symbolic versus explicit model checking have been the subject of a debate without a conclusive result. The difficulty is due to the fact that theoretical performance analysis is not possible for given problems; an optimal variable ordering for symbolic model checking is an NP-complete problem and so is computing the optimal reduction in applications of partial order reduction for explicit model checking [17]. Heuristics are used in both techniques leaving us little choice but "try and see". The difficulty of performance comparison and the importance of the empirical study on this issue is well addressed in [1].

A few comparative reports are available on this issue in different domains.

In [6], SPIN, SMV, and XMC were compared on a model of the i-protocol from GNU uucp version 1.04, showing that XMC outperforms the other two model checkers. The result confirms one of the well-known arguments that an explicit model checker performs better than a symbolic model checker on asynchronous protocol verification. Interestingly, the result has been challenged by Gerard Holzmann, the inventor of SPIN, and has been reversed by the careful optimization of the SPIN model [16]. This occasion clearly shows the difficulty of having a fair comparison between different model checkers as well as the importance of optimization, which requires high level expertise. Their recent publication [5] presents a more comprehensive comparison among various explicit model checkers in the same problem domain.

Eisner and Peled examined another well-known argument that symbolic model checking is better for hardware systems and explicit model checking is better for software systems in verifying the software of a disk controller using the symbolic model checker RULEBASE and the explicit model checker SPIN [7]. Their result shows that RULEBASE is able to model check a 2-process system with 10^{150} states, while SPIN spaces out after checking 10^8 states with 2G of memory. This result is not favorable to the argument that explicit model checking performs better for software verification, especially for communicating processes.

There also exists a performance comparison on analyzing mode confusion on a Flight Guidance System using Murϕ, SMV, and SPIN [19]. The translation is described for each model checker using an example, and the strengths and weaknesses of each tool are discussed with respect to its usability. The possibility of a state-space explosion in SPIN is also pointed out, especially because *inlining* is used for all procedures in their translation. To our best knowledge, this is the only comparative study in our domain of interest, namely, aircraft control systems. Nevertheless, the model used in the case study is quite small (several thousands of states), and, thus, the performance part is considered relatively insignificant. Since we do not have the analytical data for scalability, we cannot draw a conclusion from this small case study.

3 Background

A case study has been conducted at Rockwell-Collins in cooperation with the University of Minnesota to determine if formal methods could be used to validate system requirements at reasonable cost. A series of Flight Guidance Systems have been specified using a formal specification language RSML^{-e} (Requirements State Machine Language without events) [25], validated using the visualization and simulation facility provided by the RSML^{-e} execution

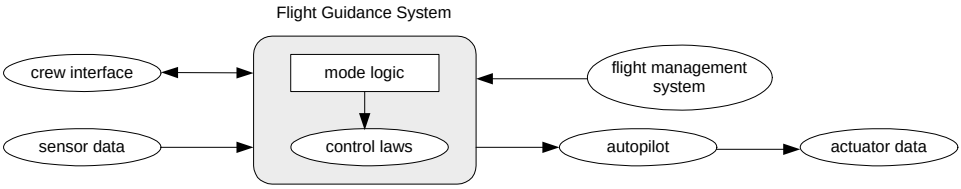


Fig. 1. Flight control System

environment NIMBUS [13], and verified with respect to several hundred functional and safety requirements using the NuSMV model checker and the theorem prover PVS [23] through fully automated translation, finding numerous errors in the model [21]. The project has been quite a success; the use of formal methods, including writing formal specifications and performing verification using NuSMV, is conducted by people at Rockwell-Collins with little help from the researchers at the University of Minnesota. The first phase of the project finished with the conclusion that "formal methods tools are maturing to the point where they can be profitably used on industrial sized problems" [21].

3.1 Flight Guidance System

A Flight Guidance System (FGS) is a component of the Flight Control System (Figure 1, borrowed from [19]). It compares the measured state of an aircraft to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. The desired state is determined by the crew interface and/or the flight management system together with the current state of the system. The guidance commands are calculated by control law algorithms selected by the mode logic. The mode logic determines which lateral and vertical modes are armed and active at any given time [20].

The FGS includes identical left and right sides where only one side is active and responsible for inputs and produces outputs. The inactive side simply copies its internal state from the active side, serving as a hot backup (Figure 2). The complex mode logic of an FGS is a representative of a class of problems frequently encountered in the design of embedded control systems. For a more detailed description of the mode logic of the FGS, please refer to [20].

The size of the FGS specification written in RSML^{-e} by people at Rockwell-Collins is around 3,500 lines (with comments) for one-sided FGS in the final stage of the project. It includes 13 input switches via crew interface, 82 enumeration variables representing the internal state of the system, and 123

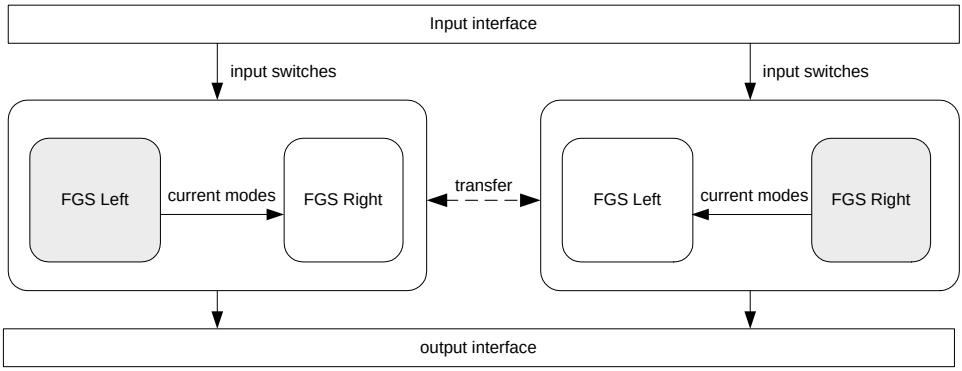


Fig. 2. Flight Guidance System with two sides

macros. Note that RSML^{-e} provides macro and function constructs to improve the readability of specifications. In the FGS specifications, macros are extensively used as an alternative representation of synchronous events which, in the end, provide a concise way to specify system properties in temporal logic [2]. Since macros encapsulate complex mode logic, the use of macros in temporal logic specification tremendously simplifies logic expression, which plays a crucial role in enhancing the usability of model checking.

A total of 298 properties are verified in batch mode using the model checker NuSMV (version 2.1) after the RSML^{-e} specification is automatically translated into the NuSMV input language. The verification time for each property varies from a couple of minutes to two hours depending on the property and on whether counter examples are generated or not. In the final stage, when all the errors in the model have been corrected, NuSMV verifies all 298 properties in about 1.5 hours on a 800MHz Linux machine with 512M of memory. The translation and the model checking process are completely automated except for the requirements properties being manually translated to temporal logic and attached to the generated NuSMV file.

3.2 Motivation of this work

The successful use of the fully automated verifier on checking several hundreds of properties is quite encouraging. Nevertheless, the size of the FGS has almost reached the limit of the NuSMV; it would require aggressive abstraction to scale up to bigger systems. In fact, verification has been focused on one side of the FGS using assumptions about the other side as invariants. These invariants are manually identified by the engineers and imposed on the NuSMV

[2] The system properties need to be specified in temporal logic [24] in order to be model checked.

model directly. As shown in Figure 3, the FGS model contains only one side that is active initially and activated/inactivated whenever the transfer switch is pressed. When active, it computes the mode of the FGS based on the current mode and the input values. When inactive, it copies the modes of the other side received as random input with some invariants.

This approach is taken because the identified requirements concern mostly the mode logic in the FGS functions that can be checked by assume-guarantee reasoning under the synchrony hypothesis[3] using one-sided FGS. The validation and verification activities have been incremental, from a very abstract FGS to a refined full-size FGS, mainly due to the lack of information on the model checking scalability. NuSMV successfully scales up to the full-size, one-sided FGS, but has not succeeded in scaling up to two-sided communicating FGSs.

From the experience, it becomes clear that symbolic model checking can be very powerful and also usable, but only up to a certain point. The technique is based on exhaustive state-space search and does not provide alternative options when it reaches its limitation. On the other hand, explicit model checking usually provides more flexibility in dealing with a large state-space, sometimes trading off exhaustiveness for efficiency. Our hope is that the flexibility of explicit model checking may be able to provide us a certain level of verification capability even for larger systems that symbolic model checking is not capable of handling. We set up two goals of the investigation to realize (or nullify) our hope; the first is to check the possibility of using the explicit model checker SPIN for verifying FGSs, and the second is to come up with a systematic approach of translating RSML^{-e} to the input language of SPIN to support full automation and better usability, if the first goal turns out to be achievable.

4 Model Checking FGSs using SPIN

We start from a translation of the RSML^{-e} model of one-sided FGSs with invariants to the input language of SPIN, Promela. Our intuitive hypothesis is that SPIN must be able to handle the synchronous one-sided FGSs in order to be scaled to the two-sided asynchronous FGSs. Our first attempt of a direct translation turns out to be too inefficient for model checking. To achieve better performance, we adopt modularization and encapsulation by utilizing the SPIN message passing mechanism and by modifying macros to cope with the structural change. The result of the change is quite promising; it

[3] The synchrony hypothesis says that the underlying machine is infinitely fast, and, hence, the reaction of the system to an input event is instantaneous [8].

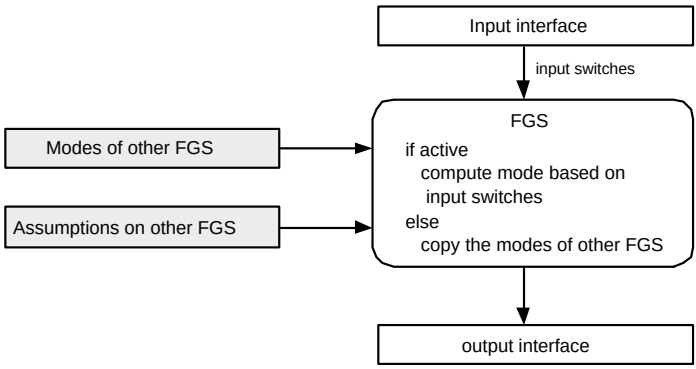


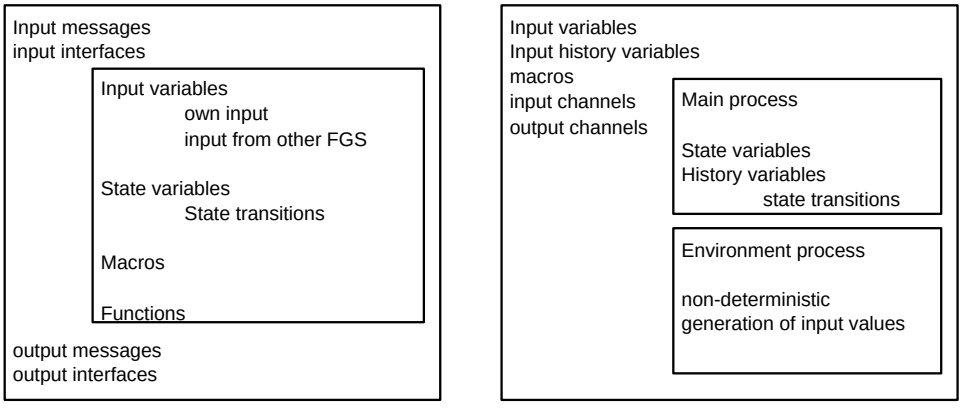
Fig. 3. Flight Guidance System with one side

enables SPIN to model check one-sided synchronous FGSs with tolerable cost in terms of time and memory compared to that of NuSMV. The same structuring and modularization approach enables SPIN to scale up to the two-sided asynchronous FGS, which is not feasible with NuSMV.

4.1 A Direct translation for model checking synchronous FGS using SPIN

RSML^{-e} [25] is a synchronous state machine language semantically similar to Lustre [11], SpecTRM-RL [18], and SCR [14]; state variables represent the current state of a system at a given time where the values of state variables are computed based on the input variable values received through input interfaces and the previous values of the state variables (system configuration). The newly computed state variable values can be sent out through output interfaces via output messages. In RSML^{-e} all variables have a global scope and an RSML^{-e} model is considered open, meaning that the system is interacting with its open environment.

Our first challenge is to model the synchronous system with a large number of variables and complex mode logic in Promela, the modeling language of SPIN. As noted in the previous section, the FGS specification includes 13 input variables, 82 state variables, and 123 macros. Furthermore, most of the macros refer to the history values of state variables, and, thus, the model needs to remember the values of the variable history. Though the depth of the history is limited to one (to the previous value), it doubles the number of variables required to keep track of the system state. In Promela, it requires at least $2 \times (13 + 82)$ variables to keep track of all the necessary values. For an initial approach, we performed a faithful translation of the RSML^{-e} specifications to Promela aiming at automated translation, as we did for the NuSMV translation. RSML^{-e} basic constructs, state variables, input variables, and macros are translated into Promela variables and C style macros



(a) Basic RSML-e construct (b) Promela translation

Fig. 4. Basic RSML^{-e} construct and its translation to Promela

as illustrated in Figure 4.

The second challenge comes from the fact that SPIN requires a closed system where all the system interaction with its environment must be explicitly specified. As shown in Figure 4, we explicitly model the system environment by introducing another process. As a result, the translated model has two processes, one for the FGS itself, the other one for the environment. The synchrony of the one-sided FGS is ensured by using the hand-shaking message passing mechanism between the environment process and the actual FGS process.

Input variables and message channels are declared as global variables in Promela, since those are to be accessed by both the FGS model and the environment model. FGS state variables as well as their corresponding history variables are translated into local variables in the FGS process, and their transition relation is translated with the Promela *if* statement. User defined enumeration types are translated to the Promela *mtype*. Macros are declared as C-style macros in Promela; SPIN performs pre-processing for C-style macros, and, thus, does not introduce additional variables for macros.

Though the basic translation is straightforward, there exist a couple of Promela-specific issues: First, the environment of the system must be explicitly specified. This means that we have to model the possible values for all 13 input variables in a non-deterministic way to simulate the random inputs from the open environment. Second, in order to model the FGS with only one side, we also have to explicitly model the possible inputs from the other FGS with constraints. Luckily, the modeling of the pure non-deterministic input is relatively simple for boolean or enumeration variables, since the Promela

if statement supports non-determinism. For example, for the heading switch HDG, the non-deterministic input value assignment can be done as follows;

```
if
:: 1 -> HDG = On;
:: 1 -> HDG = Off;
fi;
```

Nevertheless, Promela does not support imposing invariants on the model itself, and, thus, imposing constraints on the non-deterministic input values from the *other side FGS* can be quite tricky. One way is to restrict the non-deterministic value assignment with the constraints and check that the constraints are actually satisfied by the model using the assertion statement or the LTL verifier. For example, the following is an assumption to be satisfied in the input values from the *other side FGS* and the way it is imposed in NuSMV as an invariant.

- If the mode of the `other side FGS` is `On` then either `ROLL` is selected or `HDG` is selected or `NAV` is selected in the `other side FGS`.
- `INVAR Other_FGS_Mode = On` $\rightarrow$ `Other_FGS_ROLL = Selected | Other_FGS_HDG = Selected | Other_FGS_NAV = Selected`

In Promela, this invariant is manually enforced in the environment model so that the environment does not generate input values that do not obey the constraint. Figure 5 shows a small part of translated Promela code for a version of the FGS; the upper part of the left side of the figure contains samples of macro declarations, message type declarations for the values of variables, and input variable declarations. The lower part of the left side shows a part of the environment process that generates random input values with constraints. The upper part of the right side of the figure shows how state variables and their history variables are declared in the *FGS* process. Sample mode logic specifications of the *FGS* are illustrated in the lower part.

The resulting performance is very poor; SPIN quickly spaces out of the memory when checking a crucial property,

P1. Mode *annunciation* is *on* if and only if either *ROLL* is selected or *HDG* is selected or *NAV* is selected.

Verification is performed on a 800 MHz Linux machine with 768 M of memory. We set the memory limitation to 600 M in SPIN. With the hash-compact option with compression, the verification process terminates using up 625M of actual memory after 24 minutes. The maximum search depth was set at 1,000,000 and yet, SPIN reported the search depth as being too small. The situation is similar or worse with other SPIN verification options. The result

```
// macros  (total 123 of macros)
#define When_FD_Switch_Pressed  (FD_Switch == On
&& PREV_STEP_FD_Switch != On)
#define  When_Turn_FD_Off
(When_FD_Switch_Pressed_Seen && Overspeed != 1)
#define When_FD_Switch_Pressed_Seen
(When_FD_Switch_Pressed &&
No_Higher_Event_Than_FD_Switch_Pressed)
        :
// enumeration type declaration
mtype = {Off, On, Cleared, Selected, Armed, Active,
Undefined, done, LEFT, RIGHT, Disengaged, Engaged,
Capture, Track};

// input variables (total 13 of input variables,
// 13 of input history variables)
mtype FD_Switch = Undefined;
mtype HDG_Switch = Undefined;
        :
bool Offside_FD_On;
bool Offside_Modes_On;
bool Offside_Roll_Selected;
        :
proctype env(chan  input1, out){
 do
 :: 1 ->
    // initial non-deterministic value assignments
  if
   :: 1 -> FD_Switch = On;
   ::  1 -> FD_Switch = Off;
  fi;
        :
  // imposing invariants on other side FGS
  if
  :: 1 ->  Offside_Modes_On = 0;
           Offside_Roll_Selected =0;
           Offside_Hdg_Selected=0;
           Offside_Nav_Active =0;
  :: 1 ->  Offside_Modes_On = 1;
           Offside_Roll_Selected =1;
           Offside_Hdg_Selected=0;
           Offside_Nav_Active =0;
  :: 1 ->  Offside_Modes_On = 1;
           Offside_Roll_Selected =0;
           Offside_Hdg_Selected=1;
           Offside_Nav_Active =0;
  :: 1 ->  Offside_Modes_On = 1;
           Offside_Roll_Selected =0;
           Offside_Hdg_Selected=0;
           Offside_Nav_Active =1;
           Offside_Nav_Selected =1;
  fi;
        :
}
```

```
proctype FGS(chan input, out1)
{
// FGS state variables (total 82 of state variables)
mtype Onside_FD = Off;
bool Onside_FD_On = 0;
mtype Modes = Off;
mtype ROLL  = Undefined;
bool  FD_Cues_On = 0;
        :

// history variables (total 82 of history variables)
mtype PREV_STEP_Modes = Undefined;
mtype PREV_STEP_ROLL = Undefined;
        :

// mode logic
        :
  if
    :: ! Is_This_Side_Active ->
                Onside_FD = Offside_FD;
    :: (Onside_FD == Off) &&
       When_Turn_FD_On
                  ->    Onside_FD = On;
    :: (Onside_FD == On) &&
       When_Turn_FD_Off
                  ->    Onside_FD = Off;
    :: else -> skip;
  fi;

  Onside_FD_On = (Onside_FD == On);

  if
    :: Is_This_Side_Active != 1 ->
                  Modes = Offside_Modes;
    :: (Modes ==Off) &&
       When_Turn_Modes_On
       &&  Is_This_Side_Active ->
                                Modes = On;
    :: (Modes == On) &&
       When_Turn_Modes_Off
       &&  Is_This_Side_Active ->
                                Modes = Off;
    :: else -> skip;
  fi;

        :
}
```

Fig. 5. Direct translation of FGS from RSML^{-e} to Promela

is not so surprising considering the number of variables in the model and the way explicit model checking handles the system state space.

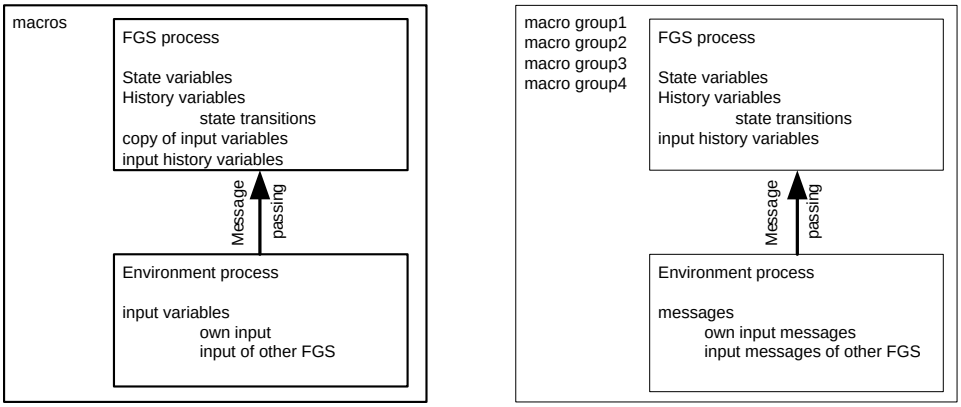
4.2 Optimization through modularization

The major source of the state-space explosion of the directly translated version of the FGS is the large number of global variables, mainly the input variables and their history variables, in addition to the large number of local state variables in the FGS process. Especially, global variables are expensive in SPIN verification since SPIN needs to keep track of all their values in the state transition graph. To optimize the performance, we look into mechanical ways to minimize the number of variables, both global and local variables. Note that one of our goals in this investigation is to support a usable verification process in practice, and, thus, any modification of the original model for the purpose of optimization needs to be systematic so that it can be automated in future work.

Conceptually, input variables can be considered either a part of the FGS state machine or a part of the environment model that generates the values of input variables. In that sense, we should be able to declare input variables as local variables in the environment process and pass them to the other process through message channels, eliminating all the global variables by modularizing each process. Unfortunately, this is not as straightforward as it sounds, since the input variables and their history values are referred by macros, which are again referred by *FGS* state transitions; *env* has to be able to update the input variables, and the *FGS* process has to be able to access those variable values via macros. For example, `When_Turn_FD_On` macro is referred in the transition condition of the state variable `Onside_FD` in the *FGS* process (Figure 5). The macro refers to the macro `When_FD_Switch_Pressed_Seen` that again refers to the macros `When_FD_Switch_Pressed` and `No_Higher_Event_Than_FD_Switch_Pressed`. These final macros refer to input variable values such as `FD_Switch`.

One possible way of dealing with such a usage of macros is to pre-process all the macros by replacing all the references to a macro in the specification with their value expression. The macros, however, are specified by practitioners in a way to match their languages used for communicating with pilots, and many of the properties (98 out of 298) to be verified are specified in terms of marcro names. Therefore, we do not want to change them if it is not really necessary [4] .

[4] We need to convert macros into variables, as we did for NuSMV translation, in order to be able to check properties specified using macro names. Nevertheless, this is out of the scope of this paper and we leave it to future work.



(a) modular approach with the original macros

(b) modular approach with modified macros

Fig. 6. Modular translation approaches

We can use the Promela message passing mechanism in order to make modularization possible with such a usage of macros; all the input variables can be declared as local variables in the *env* process, and all the values can be passed to the *FGS* process through the message channel. We separately declare input history variables as local variables in the *FGS* process in order to reduce the number of messages to be passed through the message channel. Nevertheless, in order to make use of the macros, we need to preserve the names of the passed message values, which is possible in Promela either by creating a local copy of the messages in the *FGS* process, or by creating one message channel per variable (Figure 6). Both cases are still expensive, since creating a local copy of the messages increases the number of variables, and message channels are always treated as global variables in Promela.

In order to achieve better optimization, we classify the macros according to the following four categories: (1) macros whose values are determined only by input variable values with the depth of the reference tree being one, (2) macros whose values are determined only by input variable values with the depth of the reference tree being more than one, (3) macros whose values are determined by state variables locally declared in the *FGS* process, and (4) macros whose values are determined by both input variable and state variable values. Based on this classification, we perform three types of optimization as follows.

Category 1: We change the input variable names referred in the macro to the corresponding message field names passed from the *env* process to the *FGS* process.

Category 2: We create a local variable in the *env* process for each macro

in this category so that it can be evaluated in the same process and passed to the *FGS* via a message channel.

Category 3: No change.

Category 4: We modify the names referred in the macros in this category to reflect the change of names of the input variable via a message channel and conversion of macros in category 2.

Through the approach for the macros in category 1, we do not need to keep a local copy for the input variable values, reducing the number of local variables. For example, the macro definition `When_FD_Switch_Pressed` is changed from `(FD_Switch == On) && PREV_STEP_FD_Switch != On)` to `(values.msg[0]== On && PREV_STEP_FD_Switch != On)`, since `FD_Switch` is an input variable declared in the *env* process whose values are passed to the *FGS* process through a message channel (Figure 7). *values* is the name of the message channel that passes a user-defined message type *msg*, which is an array of *mtype*. The second approach is to treat complex macros that cannot be handled by the simple approach as for category 1. For example, the `When_Turn_FD_On` macro described in the previous page has a reference tree of depth 3. We could traverse the names referenced in the reference tree and change them to reflect the changes in the first approach. Instead, we convert them into local variables in the *env* process for simplicity. In this way, all the references are resolved in the *env* module and only the resulting value will be passed to the *FGS* process (See inside *proctype env* in Figure 7). Macros in category 3 do not need to be changed, since all the macros are referenced in the *FGS* process where all the state variables are declared as local variables. The approach for macros in category 4 combines the approaches taken for the macros in category 1 and category 2. Figure 7 is a fraction of the modular translation of the FGS based on the approach, which shows the same portion of the model illustrated in Figure 5 highlighting the changed part with bigger font.

These optimization approaches enable SPIN to verify property **P1** within 20 minutes with bit-state hashing option; the search depth it has explored is 8,133, the total number of states and transitions it has explored are 2.1×10^7 and 2.9×10^7, respectively. A total of 9.938 M of memory is consumed. Partial order reduction is used by default for all our experiments. In comparison, the same property is verified using NuSMV within 10 seconds with 24 M of memory consumption.

The use of an exhaustive search algorithm, however, is still not feasible for this model. Since bitstate hashing algorithm performs a partial search of the state space, the result can be unsound.

```
// macros  (total 123 macros)
#define When_FD_Switch_Pressed
      ( values.msg[0]== On    &&
                        PREV_STEP_FD_Switch != On)

#define  When_Turn_FD_Off
                  (When_FD_Switch_Pressed_Seen &&
                                values.msg[10] != 1   )

#define When_FD_Switch_Pressed_Seen
                  (When_FD_Switch_Pressed &&
            No_Higher_Event_Than_FD_Switch_Pressed)
                              :

// enumeration type declaration
mtype = {Off, On, Cleared, Selected, Armed, Active,
Undefined, done, LEFT, RIGHT, Disengaged, Engaged,
Capture, Track};

// for input variable values and macros converted to local
// variables
typedef Array1 {
    mtype msg[33]  };

// for input variable values from the other side of FGS
typedef Array2 {
    mtype msg[13]   };

proctype env(chan out2fgs, otherinput)
{
  Array1 values;
  Array2 otherFGS;

do
 :: 1 ->
    // initial non-deterministic value assignments
   if
    :: 1 -> values.msg[0] = On;
    ::  1 -> values.msg[0] = Off;
   fi;

  // assignment of a macro value to corresponding
  // message field (category 2).

  values.message[26] = When_Turn_FD_Off;

   if
    :: 1 -> otherFGS.msg[0] = On;
    :: 1 -> otherFGS.msg[0] = Off;
   fi;
          :
}
```

```
proctype FGS (chan input_from_otherFGS,
                                    input_from_env)
{
  Array2  otherFGS;
  Array1  env;

// FGS state variables (total 82 number of state
// variables)
mtype Onside_FD = Off;
bool Onside_FD_On = 0;
mtype Modes = Off;

 input_from_env?env;

// mode logic
   if
            :: !Is_This_Side_Active  ->
                    input_from_otherFGS?otherFGS;
            :: else -> skip;
    fi;
          :
   if
    :: !Is_This_Side_Active  ->
                  Onside_FD = otherFGS.msg[0];
    :: (Onside_FD == Off) && When_Turn_FD_On ->
                                  Onside_FD = On;
    :: (Onside_FD == On) && (     env.msg[26]==1   ) ->
                                  Onside_FD = Off;
    :: else -> skip;
  fi;

  Onside_FD_On = (Onside_FD == On);

  if
    :: Is_This_Side_Active != 1 ->
                        Modes =otherFGS.msg[1];
    :: (Modes ==Off) && When_Turn_Modes_On  &&
                Is_This_Side_Active -> Modes = On;
    :: (Modes == On) && When_Turn_Modes_Off &&
                Is_This_Side_Active -> Modes = Off;
    :: else -> skip;
  fi;

          :
}
```

Fig. 7. Modular translation of FGS from RSML^{-e} to PROMELA

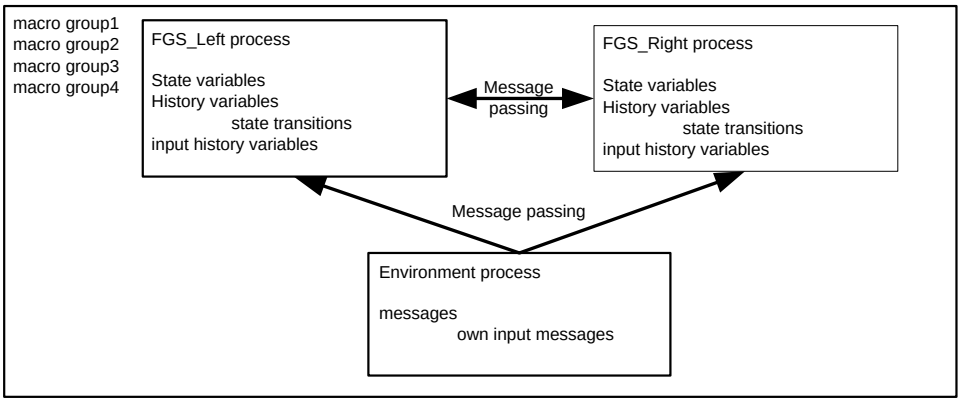


Fig. 8. Modular translation approach for communicating two FGSs

4.3 Model checking two-sided FGSs using SPIN

Once we succeed with model checking one-sided FGSs using SPIN, an extension to two-sided FGSs is fairly straightforward. We create two identical FGS processes with a communication channel between them without imposing any synchrony hypothesis. Instead of having values for the *other side FGS* generated by the environment process with some constraints, we can directly wire the two FGS processes so that a passive FGS can get the actual mode values from the other active FGS. There is no need to change the FGS process from the one-sided model, and all the random value generation and constraints for the values of *other side FGS* are removed from the *env*. With this model, SPIN successfully verifies the same property **P1** using the bit-state hashing option within 42 minutes; a total of 473 M of actual memory is consumed. The search depth has reached 486,220. The total number of states and transitions explored are 2.6×10^7 and 3.35×10^7, respectively. In comparison, NuSMV does not finish checking the same property on two-sided FGSs within two hours using the cone-of-influence reduction abstraction. Without using the abstraction, it spaces out of memory in minutes.

We would like to note that the result of SPIN verification is also based on a partial search and its coverage can be very small.

5 Discussion

We have presented our experience in using SPIN on a commercial avionics specification to answer the question that has been bothering us — which tool, the symbolic model checker NuSMV or the explicit-state based model checker SPIN, would have been more suitable for our specific problem domain in terms of their scalability and usability.

Our experiment reveals that SPIN performs poorer than NuSMV on the one-sided synchronous FGS model than NuSMV, but scales better to asynchronous two-sided FGSs once we manage to handle the one-sided FGS. Nevertheless, we do not intend to put an emphasis on the performance differences because of two major reasons; (1) the result of using SPIN can be unsound because of the use of the bit-state hashing option, and (2) the performance can be highly dependent on the level of optimization performed by the modeler. We believe that our initial optimization approach is just a starting point, and far better optimization is possible as we gain better knowledge on the model checker. We also do not rule out the possibility of using NuSMV for the two-sided FGSs by using aggressive optimization and/or abstraction in the future.

Nevertheless, we lay out three observations from this investigation; first, NuSMV appears to perform far better than SPIN on the one-sided, synchronous FGS. Second, SPIN is feasible to model check two-sided FGSs, even though it is through a partial search, which was not possible with NuSMV; SPIN may be a better choice when the size of the system is too big to use exhaustive model checking. Third, SPIN can also be usable in the sense that the optimization required in this work is systematic, and, thus, can be automated.

On the other hand, there are a number of issues to be considered; first, our experience shows that SPIN requires a more aggressive optimization approach to make it work on FGSs, and can be more sensitive to the slight differences between models. Second, SPIN's capability of handling larger systems is mainly due to the trade-off between exhaustiveness and efficiency. The bit-state hashing option allows SPIN to perform a partial search of the system with the possibility of leaving states unexplored. A couple of issues are related to the usability of the tool. The modularization and restructuring of the system model for performance improvement is based on the understanding of the techniques and implementations of SPIN. It means these verification tools still highly depend on the expertise of the user. Moreover, the result of the optimization can be difficult to understand. As we showed in Figure 7, we have changed the names of the macros, which are designed to improve readability of the specification, to incomprehensible message field names. This optimization approach may require support to help users interpret the optimized system model. Finally, SPIN supports verification of one property at a time and all optimization approaches need to be property-based. Considering that 298 properties need to be verified on the FGS routinely as the system evolves, we may need 298 different optimized models, one for each property, in the worst case. Nevertheless, we believe that property-based optimization is not a critical issue as long as we can automate the translation and the

optimization process. We leave this claim to a future investigation.

References

[1] George S. Avrunin, James C. Corbett, and Matthew B. Dwyer. Benchmarking finite-state verifiers. *Software Tools for Technology Transfer*, 2(4):317–320, 2000.

[2] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Fracesmary Modugno, David Notkin, and John D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

[3] Yunja Choi and Mats Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pages 109–118, October 2002.

[4] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[5] Y. Dong, X. Du, G.J. Holzmann, and S.A. Smolka. Fighting livelock in the GNU i-protocol: a case study in explicit-state model checking. *International Journal on Software Tools for Technology Transfer*, (4), 2003.

[6] Y. Dong and X. Du et al. Fighting livelock in the i-protocol: a comparative study of verification tools. In *Proceedings of TACAS*, 1999.

[7] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In *SPIN Workshop*, 2002.

[8] G.Berry and G.Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[9] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *17th IEEE International Conference on Automated Software Engineering*, pages 3–12, September 2002.

[10] R. Goering. Model checking expands verification's scope. *Electronic Engineering Today*, February 1997.

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[12] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java pathfinder. *International Journal on Software Tools for Technology Transfer*, pages 366–381, 2000.

[13] Mats P.E. Heimdahl, Mike Whalen, and Jeff Thompson. NIMBUS: A tool for specification centered development, September 2003. Presented at the 11th IEEE International Requirements Engineering Conference.

[14] Constance Heitmeyer, James Kirby Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.

[15] Gerard J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[16] Gerard J. Holzmann. The engineering of a model checker: The gnu i-protocol case study revisited. In *SPIN workshop*, 1999.

[17] Gerard J. Holzmann. *The* SPIN Model Checker : Primer and Reference Manual. Addison-Wesley Publishing Company, 2003.

[18] Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.

[19] Gerald Luettgen and Victor Carreno. Analyzing mode confusion via model checking. In *SPIN workshop*, 1999.

[20] Steve Miller and Alan Tribble. A methodology for improving mode awareness in flight guidance design. In *Digital Avionics Systems Conference*, 2002.

[21] Steven P. Miller, Alan C. Tribble, and Mats P.E. Heimdahl. Proving the shalls. In *Formal Methods Europe*, 2003.

[22] NuSMV: A New Symbolic Model Checking. Available at http://nusmv.irst.itc.it/.

[23] S. Owre, N. Shankar, and J.M. Rushby. *User Guide for the PVS Specification and Verification System*. Computer Science Laboratory; SRI International, Menlo Park, CA 94025, beta release edition, March 1993.

[24] A. Pnueli. The temporal logic of programs. In *Proc. 18th IEEE Symp. Foundations of Computer Science*, pages 46–57, 1977.

[25] Michael W. Whalen. A formal semantics for RSML^{-e}. Master's thesis, University of Minnesota, May 2000.